Chapter 13

Using, Evaluating, and Implementing the Optimal and Stack Instruction sets

Let’s see how we can multiply on H1 (there are several ways)

FIGURE 13.1

```
int mult_rec(int x, int y)
{
    if (y == 0)
        return 0;
    return mult_rec(x, y-1) + x;
}
```

FIGURE 13.2

```
int mult_loop(int x, int y)
{
    int product = 0;

    while (y > 0) {
        product = product + x;
        y--;
    }
    return product;
}
```

Shift-add algorithm

FIGURE 13.3

```
    0010     multiplicand
    0101     multiplier
    ----
    0010
   0000      partial products
  0010
 0000
--------
00001010     product
```

FIGURE 13.4

```
set product to 0

if multiplier == 0, exit
if rightmost bit of multiplier == 1,
   add multiplicand to product
shift the multiplier right one position
shift the multiplicand left one position
```

Implementation of the shift-add algorithm in C++

FIGURE 13.5

```
int mult_shift_add(int multiplicand, unsigned multiplier)
{
  int product = 0;
  while (multiplier != 0) {
    if (multiplier & 1 == 1)                  // rightmost bit == 1?
       product = product + multiplicand;
    multiplier = multiplier >> 1;             // logical shift right
    multiplicand = multiplicand << 1;         // shift left
  }

  return product;
}
```

Calling sequence for the function on the preceding slide

```
ld   multiplier
push
ld   multiplicand
push
call mult_shift_add
dloc 2
```

Better way to determine lsb of multiplier—use **sodd** not **and**

```
if (multiplier & 1 == 1)
```

more efficiently with the following sequence:

```
ldr  3      ; get multiplier
sodd        ; skip next instruction if multiplier is odd
ja   @L2
```

FIGURE 13.6

```
        !o        ; directive to use optimal inst set
@mult_shift_add$iui;
        esba

        ; int product = 0;
        ldc   0
        push            ; allocate and initialize product to 0

        ; while (multiplier != 0) {
@L0:    ldr   3         ; get multiplier
        jz    @L1

        ; if (multiplier & 1 == 1)        // rightmost bit == 1?
        ldc   1
        push
        ldr   3         ; get multiplier
        and
        jz    @L2

        ; product = product + multiplicand;
        ldr   -1        ; get product
        addr 2          ; add shifted multiplicand
        str   -1        ; save produce
@L2:
        ; multiplier = multiplier >> 1;
        ldr   3         ; get multiplier
        shrl 1
        str   3         ; save multiplier

        ; multiplicand = multiplicand << 1;
        ldr   2         ; get multiplicand
        shll 1          ; shift multiplicand left 1 position
        str   2         ; save multiplicand
        ja    @L0
@L1:
        ; return product;
        ldr   -1        ; load product into ac register
        reba
        ret
        public @multi_shift_add$iui
```

Levels at which multiplication can be performed

- At the C++ level (with **`mult_rec`**, **`mult_loop`**, or the C++ version of **`mult_shift_add`**)
- At the assembly level (with a **`mult_shift_add`** version written directly in assembly language)
- At the microlevel (with the `m` instruction)
- At the circuit level (with the `mult` instruction)

Determining microinstruction counts

FIGURE 13.7

```
Starting session.  Enter h or ? for help.
---- [T7]  0: ld   /0 004/ enable
Microlevel enabled
---- [T7]  0: ld   /0 004/ g
  0: ld   /0 004/ ac=0000/0002
  1: add  /2 005/ ac=0002/0005
  2: st   /1 006/ m[006]=0000/0005
  3: halt /FFFF /
Machine inst count =     4 (hex) =        4 (dec)
Micro   inst count =    21 (hex) =       33 (dec)
---- [T7] q
```

Use s command to get microinstruction count

FIGURE 13.8

```
Starting session.  Enter h or ? for help.
---- [T7] 0: ld   /0 004/ g
  0: ld   /0 004/ ac=0000/0002
  1: add  /2 005/ ac=0002/0005
  2: st   /1 006/ m[006]=0000/0005
  3: halt /FFFF /
Machine inst count =       4 (hex) =      4 (dec)
---- [T7] s
Machinecode file   = sum.mac          Size = 7    (hex) =    7 (dec)
Microcode file     = none             Size = 93   (hex) =  147 (dec)
Config file        = none
Log file (on)      = sum.log
Answer file        = none
Simulation mode    = horizontal
Microlevel         = disabled
Shifter            = one-position
Display mode       = machine-level
Cmd line addr      = F3C    (hex) =   3900 (dec)
Load point         = 0      (hex) =      0 (dec)
Machine inst count = 4      (hex) =      4 (dec)
Micro   inst count = 21     (hex) =     33 (dec)
---- [T7] q
```

Using mc to get microinstruction counts

FIGURE 13.9

```
Starting session.  Enter h or ? for help.
---- [T7] 0: ld   /0 004/ mc
Machine-level display mode + counts
---- [T7] 0: ld   /0 004/ g
  0: ld   /0 004/ ac=0000/0002   11t
  1: add  /2 005/ ac=0002/0005   11t
  2: st   /1 006/ m[006]=0000/0005   9t
  3: halt /FFFF /   2t
Machine inst count =      4 (hex) =      4 (dec)
---- [T7] mc-
Machine-level display mode
---- [T7] q
```

Program to time the m instruction

FIGURE 13.10 testm.mas

```
              ; multiply 3000 x 4
!o            ; configuration/microcode file to use
ldc  3000     ; get multiplier
push          ; push one number onto stack
ldc  4        ; get multiplicand
m             ; multiply using shift-add in microcode
halt
```

m uses 75 microinstructions—a lot more than ldc

FIGURE 13.11

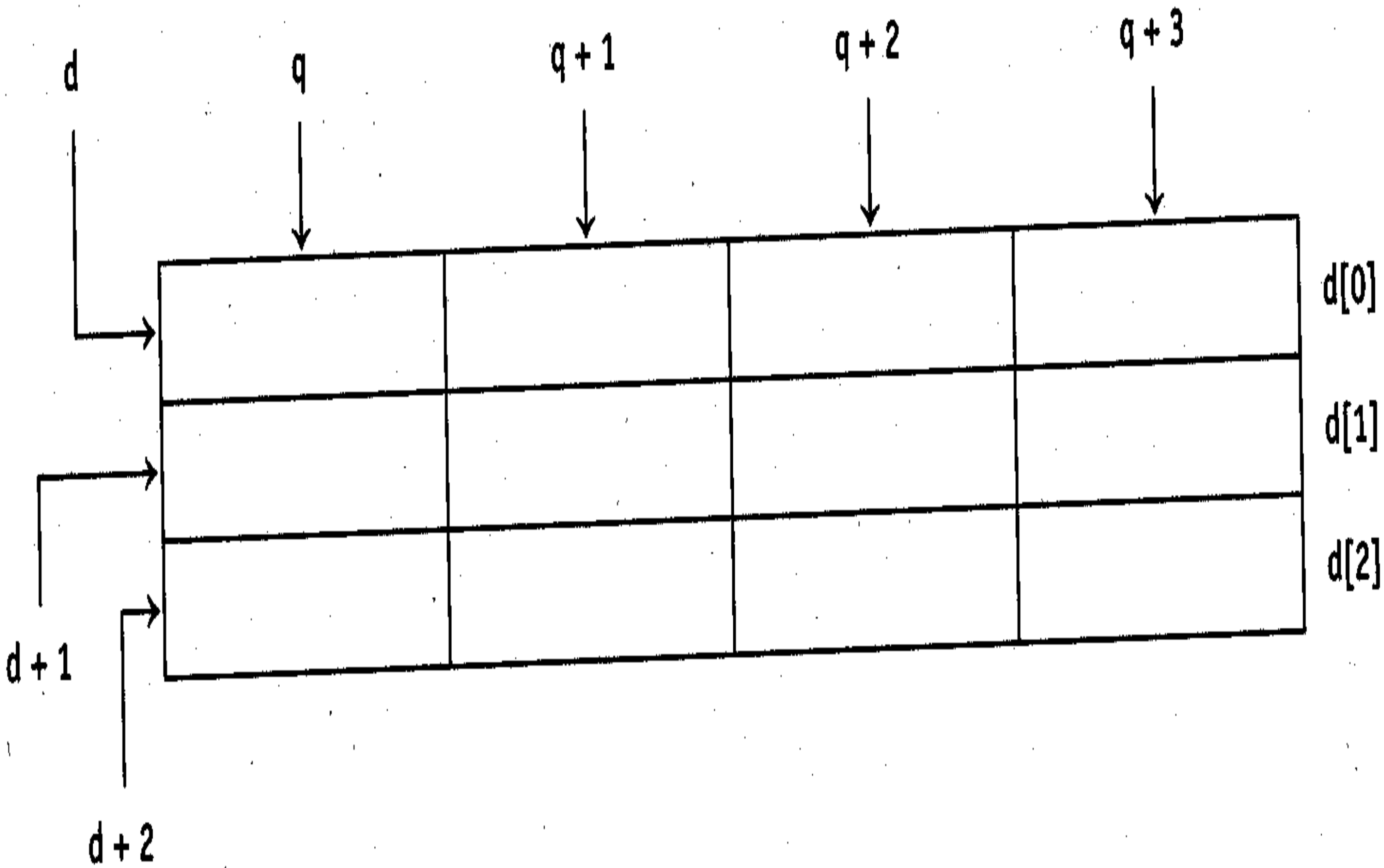
```
Reading configuration file o.cfg
Reading microcode file o.hor

Starting session.  Enter h or ? for help.
---- [T7] 0: ldc  /8 BB8/ mc
Machine-level display mode + counts
---- [T7] 0: ldc  /8 BB8/ g
  0: ldc  /8 BB8/ ac=0000/0BB8   8t
  1: push /F3 00/ m[FFF]=0000/0BB8  sp=0000/FFFF   13t
  2: ldc  /8 004/ ac=0BB8/0004   8t
  3: m    /FF3 0/ sp=FFFF/0000  ac=0004/2EE0   75t
  4: halt /FFFF /  2t
Machine inst count =     5 (hex) =      5 (dec)
---- [T7] q
```

Two-dimensional arrays

Think of a two dimensional array as a one-dimension array in which each slot is itself an array

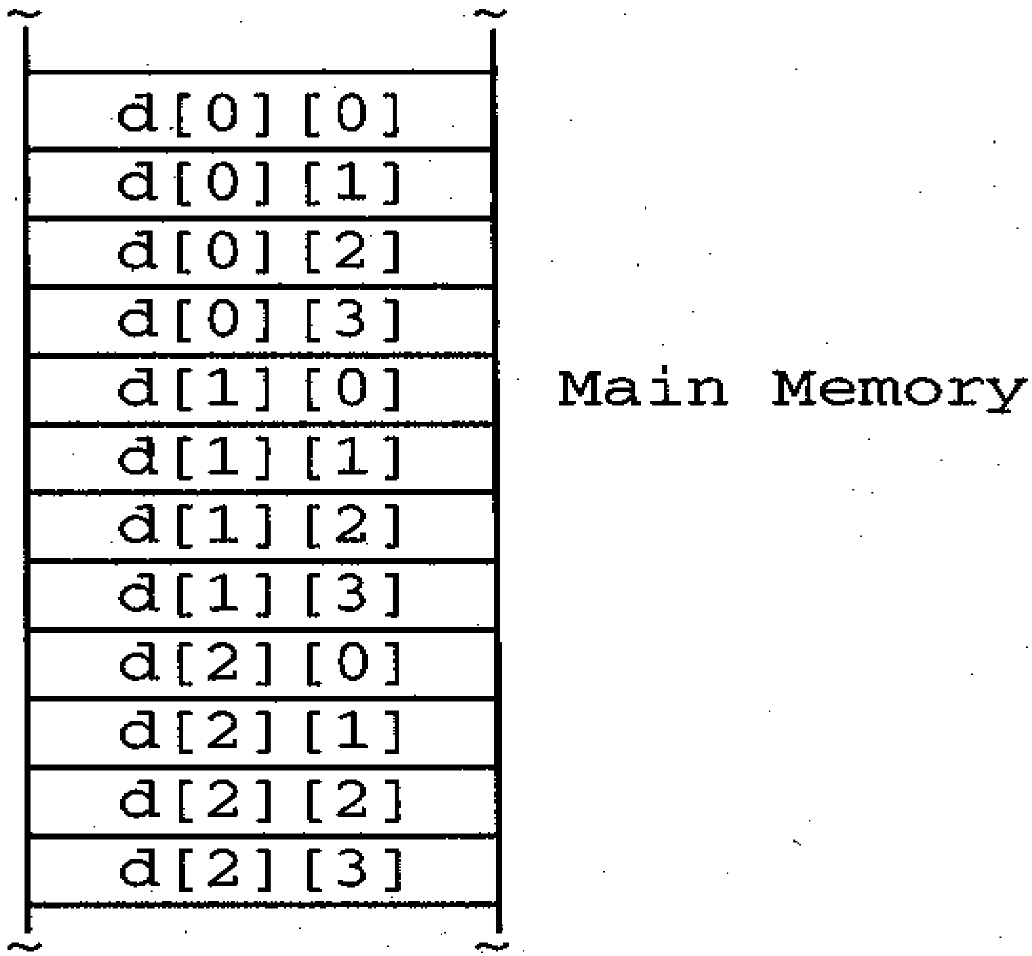
```
int d[3][4];

int *q = &d[0][0];
```

FIGURE 13.12



Row-major order

FIGURE 13.13



The location of d[2][3] is determined at compile time

```
int d[3][4];        // global variable
d[2][3] = 99;
```

is translated to

```
ldc 99
st   d + 11

d: dw   12 dup 0
```

Location of d[i][j] must be determined at run time (see next slide). This computation needs the second dimension's size but not the first dimension's size.

```
int d[3][4];
d[i][j] = 99;
```

Address of d[i][j] given by
(address of d) + i x 4 + j

FIGURE 13.14

```
; push right-hand side of assignment statement
ldc  99
push

; calculate i x 4 (number of words preceding ith row)
ld    i
push
ldc   4 ; get size of second dimension
mult

; get offset to d[i][j] from beginning of array
add   j ; add second index to get offset of d[i][j]
push    ; save it on top of stack

; get address of the beginning of d
ldc   d

; get address of d[i][j]
addr -1 ; add top of stack (assume rel add is -1)
dloc  1 ; remove top of stack

sti     ; pop 99 into d[i][j]
```

```
int d[3][4];
p = d;
```

What should be the type of p? d points to the *first row* of the d array. This first row is an int array with 4 slots.

```
int (*p)[4]
```

FIGURE 13.15

```
void f(int (*p)[4])
{
    int i = 1, j = 2;
    p[i][j] = 99;
}
int main()
{
    int d[3][4];
    f(d);
    return 0;
}
```

void f(int (*p)[4])

and

void f(int p[][4])

are equivalent. Read “[]” as “pointer to”

C++ structs, classes and objects

Let’s implement a program, first with structs, then with classes. What is the output of the program on the next slide?

FIGURE 13.16

```
#include <iostream>
using namespace std;

struct Coordinates {
    int x;
    int y;
};
void set(Coordinates *p, int a, int b)
{
    p -> x = a;
    p -> y = b;
}
void display(Coordinates *p)
{
    cout << "x = " << p -> x << endl;
    cout << "y = " << p -> y << endl;
}
int main()
{
    Coordinates c1;
    Coordinates c2;
    set(&c1, 1, 2);          // access c1 through set function
    set(&c2, 3, 4);          // access c2 through set function
    display(&c1);            // access c1 through display function
    display(&c2);            // access c2 through display function
    c1.y = 22;               // access c1 directly
    display(&c1);            // access c1 through display function
    return 0;
}
```

Output from program on preceding slide is

```
x = 1
y = 2
x = 3
y = 4
x = 1
y = 22
```

The assembly code for this program is on the next three slides.

FIGURE 13.17

```
          !o
@set$p11Coordinatesii:
          esba

                              ; p -> x = a;
          ldr   3                 ; get a
          push                    ; push a
          ldr   2                 ; get p
          sti                     ; pop a to location p points to

                              ; p -> y = b;
          ldr   4                 ; get b
          push                    ; push b
          ldr   2                 ; get p
          addc  1                 ; get p + 1
          sti                     ; pop b to location  p + 1 points to

          reba
          ret
;===========================================================
@display$p11Coordinates:
          esba

                              ; cout << "x = " << p -> x << endl;
          ldc   @m0               ; get address of @m0
          sout                    ; display string
          ldr   2                 ; get p
          ldi                     ; get *p
          dout                    ; display it
          ldc   '\n'              ; endl
          aout

                              ; cout << "y = " << p -> y << endl;
```

(continued)

FIGURE 13.17
(continued)

```
           ldc    @m1        ; get address of @m1
           sout              ; display string
           ldr    2          ; get p
           addc 1            ; get p + 1 (address of p -> y)
           ldi               ; get *(p + 1)
           dout              ; display it
           ldc    '\n'       ; endl
           aout

           reba
           ret
;=============================================================
main:      esba

           aloc 2            ; Coordinates c1;

           aloc 2            ; Coordinates c2;

                             ; set(&c1, 1, 2);
           ldc    2          ; get constant 2
           push              ; push it
           ldc    1          ; get constant 1
           push              ; push it
           ldc    -2         ; get relative address of c1
           cora              ; convert to absolute address
           push              ; push absolute address of c1
           call @set$p11Coordinatesii
           dloc 3            ; remove parameters from stack

                         ; set(&c2, 3, 4);
           ldc    4          ; get constant 4
           push              ; push it
           ldc    3          ; get constant 3
           push              ; push it
           ldc    -4         ; get relative address of c2
           cora              ; convert to absolute address
           push              ; push absolute address of c2
           call @set$p11Coordinatesii
           dloc 3            ; remove parameters from stack

                         ; display(&c1);
           ldc    -2         ; get relative address of c1
           cora              ; convert to absolute address
           push              ; push absolute address of c1
```

(continued)

FIGURE 13.17 (continued)

```
        call @display$p11Coordinates
        dloc 1          ; remove parameter from stack

                    ; display(&c2);
        ldc  -4         ; get relative address of c2
        cora            ; convert to relative address
        push            ; push absolute address of c2
        call @display$p11Coordinates
        dloc 1          ; remove parameter from stack

                    ; c1.y = 22;
        ldc  22         ; get constant 22
        push            ; push it
        ldc  -1         ; get relative address of c1.y
        cora            ; convert to absolute address
        sti             ; pop 22 into c1.y

                    ; display(&c1);
        ldc  -2         ; get relative address of c1
        cora            ; convert to absolute address
        push            ; push absolute address of c1
        call @display$p11Coordinates
        dloc 1          ; remove parameter from stack

        ldc  0      ; return 0;
        reba
        ret
;============================================================
@m0:    dw    "x = "
@m1:    dw    "y = "
        public @set$p11Coordinatesii
        public @display$p11Coordinates
        public main
```

Let’s rewrite our C++ struct program so that it uses reference parameters instead of pointers. The assembly code is the same as the pointer version, except for name mangling.

FIGURE 13.18

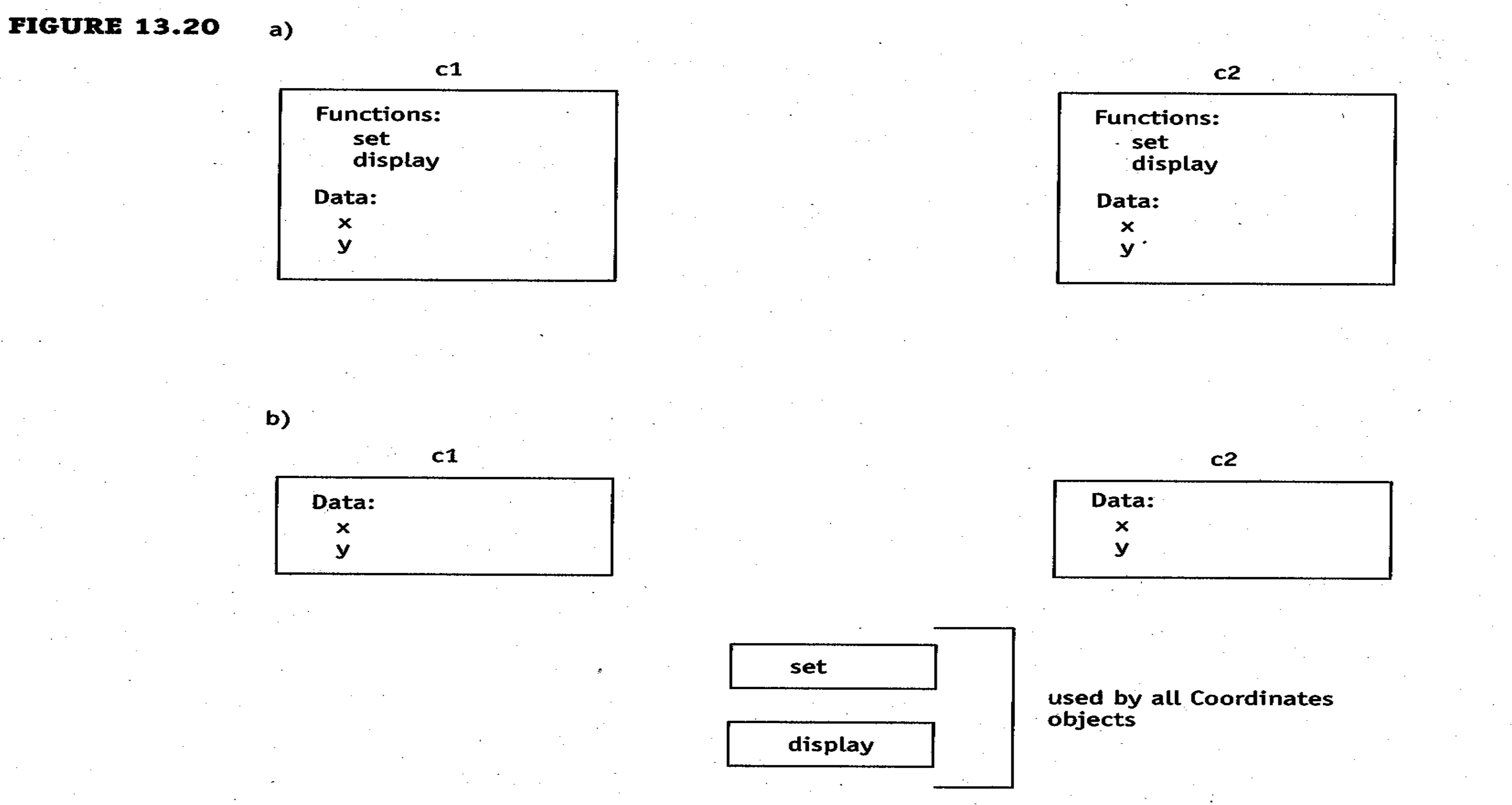
```
#include <iostream>
using namespace std;

struct Coordinates {
    int x;
    int y;
};
void set(Coordinates &c, int a, int b)
{
    c.x = a;
    c.y = b;
}
void display(Coordinates &c)
{
    cout << "x = " << c.x << endl;
    cout << "y = " << c.y << endl;
}
int main()
{
    Coordinates c1, c2;
    set(c1, 1, 2);         // access c1 through set function
    set(c2, 3, 4);         // access c2 through set function
    display(c1);           // access c1 through display function
    display(c2);           // access c2 through display function
    c1.y = 22;             // access c1 directly
    display(c1);           // access c1 through display function
    return 0;
}
```

Now let's rewrite our program using a class

FIGURE 13.19

```
#include <iostream>
using namespace std;

class  Coordinates {
   public:
      void set(int a, int b);                  // function prototype
      void display();                          // function prototype
   private:
      int x;
      int y;
};
void Coordinates::set(int a, int b)            // function definition
{
   x = a;
   y = b;
}
void Coordinates::display()                    // function definition
{
   cout << "x = " <<  x << endl;
   cout << "y = " <<  y << endl;
}
int main()
{
   Coordinates c1, c2;
   c1.set(1, 2);            // access c1 through set function
   c2.set(3, 4);            // access c2 through set function
   c1.display();            // access c1 through display function
   c2.display();            // access c2 through display function
// c1.y = 22;               // illegal--y is private
   return 0;
}
```

An object conceptually, and in reality



The statement

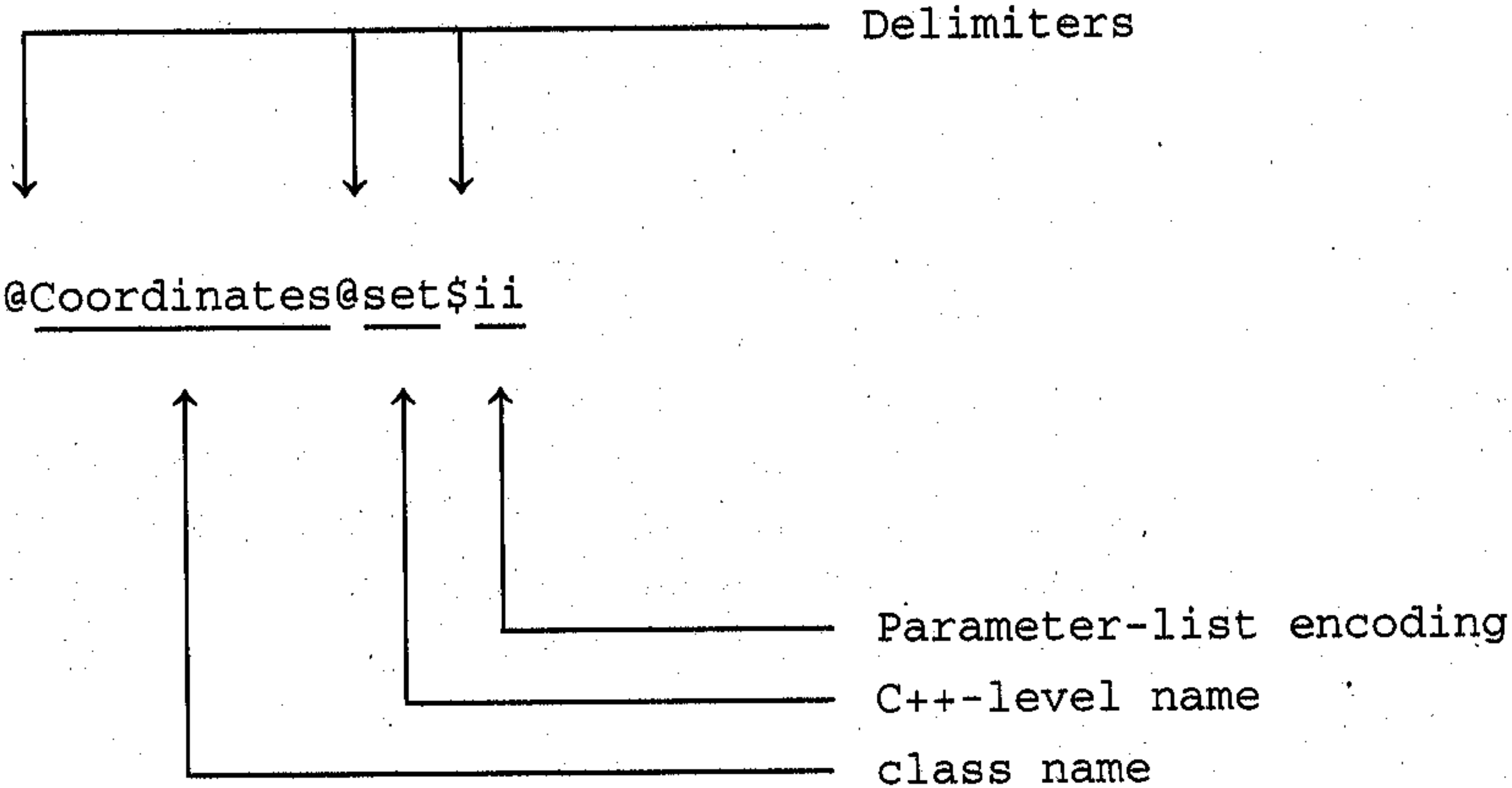
```
c1.set(1,2);
```

passes 1 and 2 to **set**. But it also passes the address of **c1**'s data area. In fact, this statement is translated exactly like

```
set(&c1, 1, 2);
```

in our first struct program.

Function mangling with classes



Because **set** is passed the address of the data area for the **c1** object, it must dereference this address to access **c1**’s data (just like the two preceding struct programs). The next slide shows the code in **set** that accesses the **x** data field.

FIGURE 13.21

```
     !o
@Coordinates@set$ii:
          esba

                        ; x = a;
          ldr   3          ; get a
          push             ; push it
          ldr   2          ; get address of x
          sti              ; pop a into x

```

(continued)

The assembly code for all three programs—the two struct programs and the class program—is the same except for name mangling. The OO version hides the address passing and dereferencing, much like reference parameters hide the same.

Inheritance

A mechanism of creating a new class from an already existing class. The original class is called the *base class.* The new class is called the *derived class.*

class B is derived from class A

```
class A {...};
class B: public A {...};
```

The **B** class ***redefines*** the **set** function because it contains a **set** function with the same signature as the **set** function in class **A.** The complete program is on the next two slides.

Program using inheritance

FIGURE 13.22

```
#include <iostream>
using namespace std;

class A {                    // A is the base class
   public:
```

(continued)

FIGURE 13.22
(continued)

```
        void set();
        void display_x();
    protected:              // allows derived class to access x
        int x;
};
void A::set()
{
    x = 1;
}
void A::display_x()
{
    cout << x << endl;
}
class B: public A {     // B is derived from A
    public:
        void set();         // redefines set() from A
        void display_y();
    private:
        int y;
};
void B::set()               // sets both x and y
{
    x = 2;                  // illegal if x is private
    y = 3;
}
void B::display_y()
{
    cout << y << endl;
}
int main()
{
    A a;
    a.set();
    a.display_x();          // displays 1
//  a.display_y();          // illegal--display_y not in a
    B b;
    b.set();                // invoke set from B
    b.display_x();          // displays 2
    b.display_y();          // displays 3
    b.A::set();             // explicitly invoke set in from A
    b.display_x();          // displays 1
//  a.B::set();             // illegal--B-level set not in a
    return 0;
}
```

a is an **A**-level object; **b** is a **B**-level object

FIGURE 13.23 a)

a

Functions:
- set
- display_x

Data:
- x

b)

b

Functions:
- set (from A)
- set (from B)
- display_x (from A)
- display_y (from B)

Data:
- x (from A)
- y (from B)

Calling the one and only **set** function in object **a**

```
a.set();
```

the compiler generates the code

```
ldc  -1            ; get relative address of a
cora               ; convert to absolute address
push
call @A@set$v      ; call set function in A
dloc 1             ; remove parameter from stack
```

Calling the B-level **set** function in the object **b**

```
b.set();
```

the compiler generates a similar sequence, but calls **@B@set$v** (the **B**-level set function) because it knows that the class **B** has its own **set** function that redefines the **set** function from class **A**:

```
ldc  -3          ; get relative address of b
cora             ; convert to absolute address
push
call @B@set$v
dloc 1           ; remove parameter from stack
```

Can still call the A-level **set** function in object **b**

```
b.A::set();
```

we have qualified the **set** function name with the class name **A** using the scope resolution operator (**::**). Accordingly, the compiler generates the following code that calls the **A**-level **set** function:

```
ldc  -3          ; get relative address of b
cora             ; convert to absolute address
push
call @A@set$v
dloc 1           ; remove parameter from stack
```

An **A**-level pointer can point to either an **A-level** object or a **B** object (it can "point across or down").

A **B**-level pointer can point to a **B**-level object. But it cannot "point up" to an **A**-level object.

Why does this make sense? For the answer, see the next slide.

```
A a;
B b;
A* aptr;
B* bptr;
aptr = &a;    // ok, points across
aptr = &b;    // ok, points down
bptr = &b;    // ok, points across
bptr = &a;   // illegal, cannot point up
```

A **B** object is an **A** object with some extra stuff but an **A** object is not a **B** object.

FIGURE 13.25 a) toolbox <-> class A

```
Tools
```

b) toolbox with a cheese sandwich <-> class B

```
Tools
Cheese sandwich
```

A program illustrating the use of inheritance and object pointers is on the next two slides.

FIGURE 13.26

```
#include <iostream>
using namespace std;

class A {                           // A is the base class
   public:
      void set();
      void display_x();
   protected:                       // allows derived class to access x
      int x;
};
void A::set()
{
   x = 1;
}
void A::display_x()
{
   cout << x << endl;
}
class B: public A {                 // B is derived from A
   public:
      void set();                   // redefines set() from A
      void display_y();
   private:
      int y;
};
void B::set()                       // sets both x and y
{
   x = 2;                           // illegal if x is private
   y = 3;
}
void B::display_y()
{
   cout << y << endl;
}
int main()
{
   A a;
   A* aptr;
   B b;
   B* bptr;
```

(continued)

FIGURE 13.26 (continued)

```

    aptr = &a;
    aptr -> set();          // invokes set in A
    aptr -> display_x();    // invokes display_x in A--displays 1

    aptr = &b;
    aptr -> set();          // invokes set in A
    aptr -> display_x();    // invokes display_x in A--displays 1

    bptr = &b;
    bptr -> set();          // invokes set in B
    bptr -> display_x();    // invokes display_x in A--displays 2
    bptr -> display_y();    // invokes display_y in B--displays 3

    bptr ->A::set();        // invokes set in A
    bptr -> display_x();    // invokes display_x in A--displays 1

//  bptr = &a;              // illegal--bptr cannot point "up"
    return 0;
}
```

aptr -> set();

calls the **A**-level **set** function, @A@set$v, even if **aptr** points down to a **B**-level object. It is the level of the pointer and not the level of the object pointed to that determines which set function (**A**-level or **B**-level) is called.

But if **set** is a *virtual* function, then it is the level of the object—not the level of the pointer—that determines which set function is called.

Virtual functions are difficult to implement. How does the compiler know at *compile time* what a pointer points to at *run time*? (Answer: it doesn't)

```
cin >> x;
if (x == 1)
     aptr = &a;
else
     aptr = &b;
```

precedes the statement,

```
aptr -> set();
```

Then, it would be impossible for the compiler to determine what **aptr** would contain at run time because the contents of **aptr** depend on the value input into **x** at run time. A further complication is illustrated by the following loop:

```
for (int i = 1; i <= 10; i++) {
   if (i % 2 == 0)
      aptr = &a;
   else
      aptr = &b;

   // aptr points to a when i = 2, 4, 6, 8, 10
   // aptr points to b when i = 1, 3, 5, 7, 9
   aptr -> set();
}
```

Adding the keyword **virtual** to the definition of the **set** function in class **A** makes both **set** functions virtual:

```
class A {
   public:
      virtual void set();
      ...
};
```

Adding the keyword **virtual** to the definition of the **set** function affects

aptr ->set();

when **aptr** points to a **B**-level object. With **virtual**, this call now invokes the **B**-level set function. Without **virtual**, it invokes the **A**-level set function.

Without the `virtual` keyword on line 6, the program in Figure 13.26 displays

```
1
1
2
3
1
```

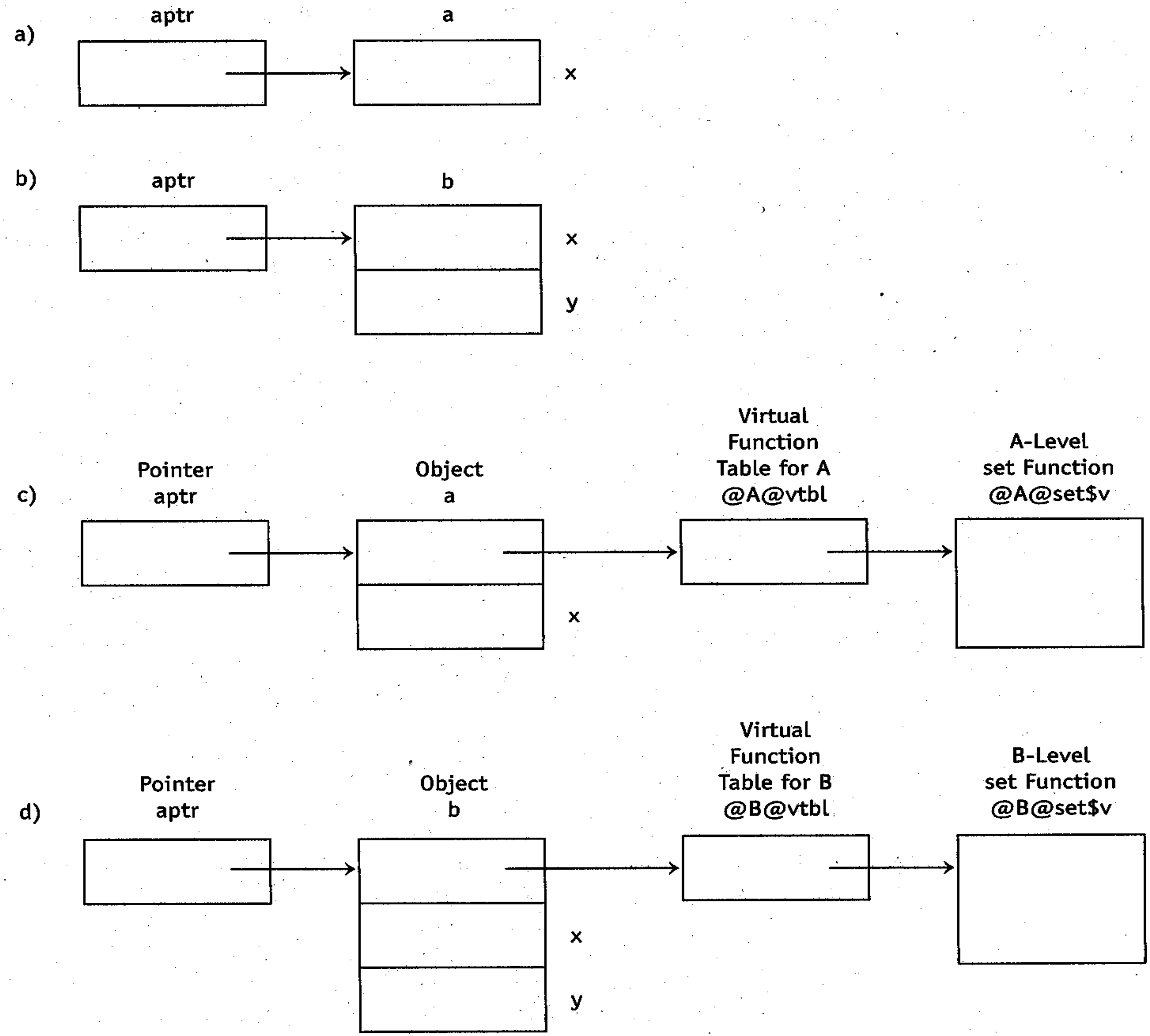
With the `virtual` keyword, it displays

```
1
2
2
3
1
```

With **virtual**, **aptr** points to the object’s data which includes a pointer to a *virtual function table* which in turn points to the correct **set** function to execute.

See the next slide.

FIGURE 13.28


a)
aptr
a
x
b)
aptr
b
x
y
c)
Pointer
aptr
Object
a
x
Virtual
Function
Table for A
@A@vtbl
A-Level
set Function
@A@set$v
d)
Pointer
aptr
Object
b
x
y
Virtual
Function
Table for B
@B@vtbl
B-Level
set Function
@B@set$v

Code for non-virtual aptr -> set();

```
ldr  -2          ; get aptr (address of object)
push             ; create parameter
call @A@set$v    ; A-level set function
dloc 1           ; remove parameter
```

Code for virtual aptr -> set();

```
ldr   -2       ; get aptr
push           ; create parameter
ldi            ; get ptr to virtual function table
ldi            ; get address of virtual function
cali           ; call virtual function
dloc 1         ; remove parameter
```

Using object's name to call virtual function

```
a.set();
```

generates the code

```
ldc  -2           ; get relative address of a
cora              ; convert to absolute address
push              ; create parameter
call @A@set$v
dloc 1            ; remove parameter
```

whether or not the `set` function is virtual. Similarly, the call

```
b.set();
```

generates

```
ldc  -6           ; get relative address of b
cora              ; convert to absolute address
push              ; create parameter
call @B@set$v
dloc 1            ; remove parameter
```

```
aptr -> set();
```

is a single item that can have multiple effects (depending on the level of the object that aptr is pointing to. This “single item/multiple effects” feature is called *polymorphism.*

Function overloading is also an example of polymorphism.

```
fol(10);
fol();
fol(2, 3);
```

Call by name

- Rolls-Royce of parameter-passing mechanisms.
- Too costly to be practical (not supported by most modern programming languages).
- But elegant and worthy of our attention.
- Its use can make programs hard to understand.
- Hard to implement.
- Involves a lot of run-time overhead.
- ‘#’ flags name parameter.

With call by name, the argument—not its value or address—replaces the corresponding parameter. On the call

```
n(p + q);
```

p + q replaces its parameter **x** in the **n** function. **p + q** is evaluated ***every time x is used***.

What is the output for the program on the next slide?

Use ‘#’ to specify a name parameter

FIGURE 13.30

```
#include <iostream>
using namespace std;

int p = 1;
void n(int #x)                    // # signals the name mechanism
{
        int y;

        y = x;
        cout << y << endl;
        p = p + 5;
        y = x;
        cout << y << endl;
}

int main()
{
        int q = 2;
        n(p + q);                 // argument is p + q
        n(q);                     // argument is q
        return 0;
}
```

The output of the program on the preceding slide is

3

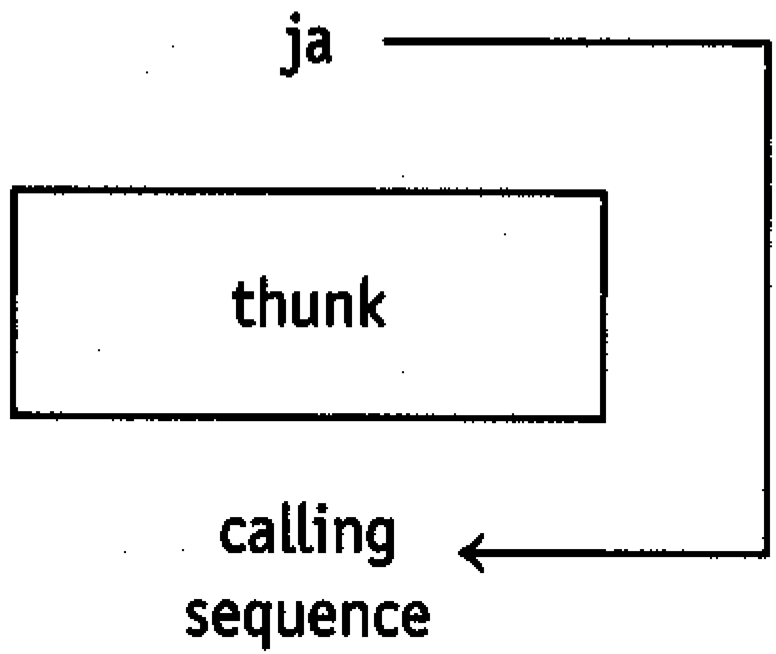
8

2

2

Call by name is implemented with *thunks.* A thunk is code that evaluates the argument.

FIGURE 13.31



Assembly code for n(p+q);

```
       aloc   1        ; create implicit var (rel add = -2)
       ja     @L0      ; jump over thunk
@L1:   ld     p        ; get p
       addr   -1       ; add q
       str    -2       ; store sum in implicit variable
       ldc    -2       ; get rel add of implicit variable
       cora            ; convert to absolute address
       ret             ; end of thunk

@L0:   ldc    @L1      ; get address of thunk
       push            ; create parameter
       call   @n$ni    ; call n
       dloc   2        ; remove parm and imp var
```

Assembly code for y = x; in **n** function

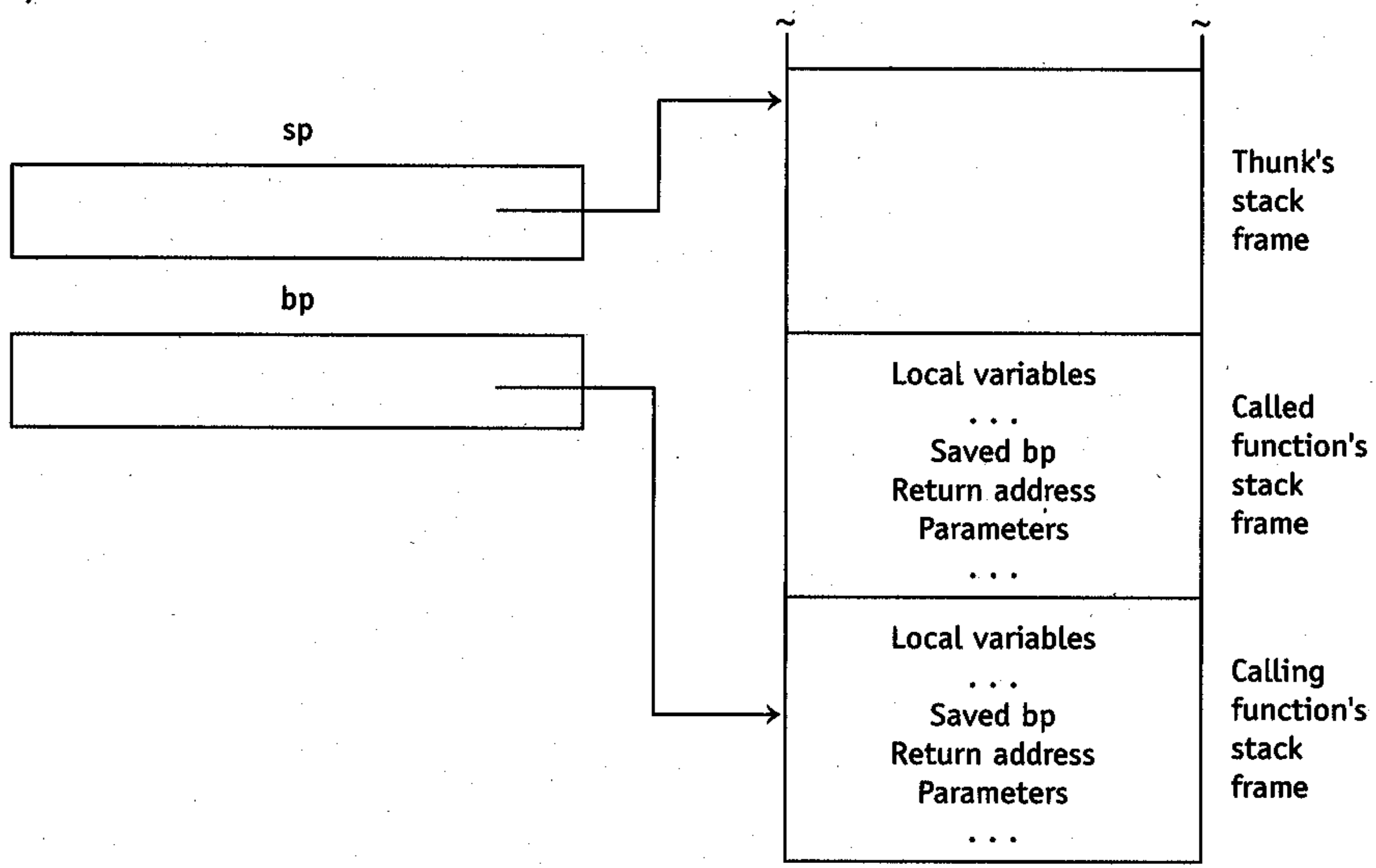
```
ldr  2       ; get x (address of thunk)
pbp          ; save bp on stack
bpbp         ; restore thunk's bp
cali         ; call thunk
pobp         ; restore bp

ldi          ; get val computed by thunk
str   -1     ; store in y
```

When n is called, bp is set to point to the stack frame of n. When n calls the thunk, it resets bp to point to the caller's stack frame. Thus, the thunk executes in the calling function's *environment.*

bp when thunk is executing

FIGURE 13.33 (continued)

c)



Thunk must return the address of value computed.

FIGURE 13.34

```
int t = 10, q;
void name_test(int #x)
{
     q = x;   // name parameter is on right side
     x = 5;   // name parameter is on left side
}
int main()
{
     name_test(t);
     return 0;
}
```

Stack Instruction Set

A completely new instruction set in which the top of the stack assumes the role that the ac register has in the optimal instruction set.

Load instructions in the optimal instruction set

```
ld    100     ; load from location 100
ld    x       ; load from location x
ldr   -2      ; load relative from relative address -2
ldc   5       ; load constant 5
ldc   x       ; load address of x
```

Push instructions in the stack instruction set

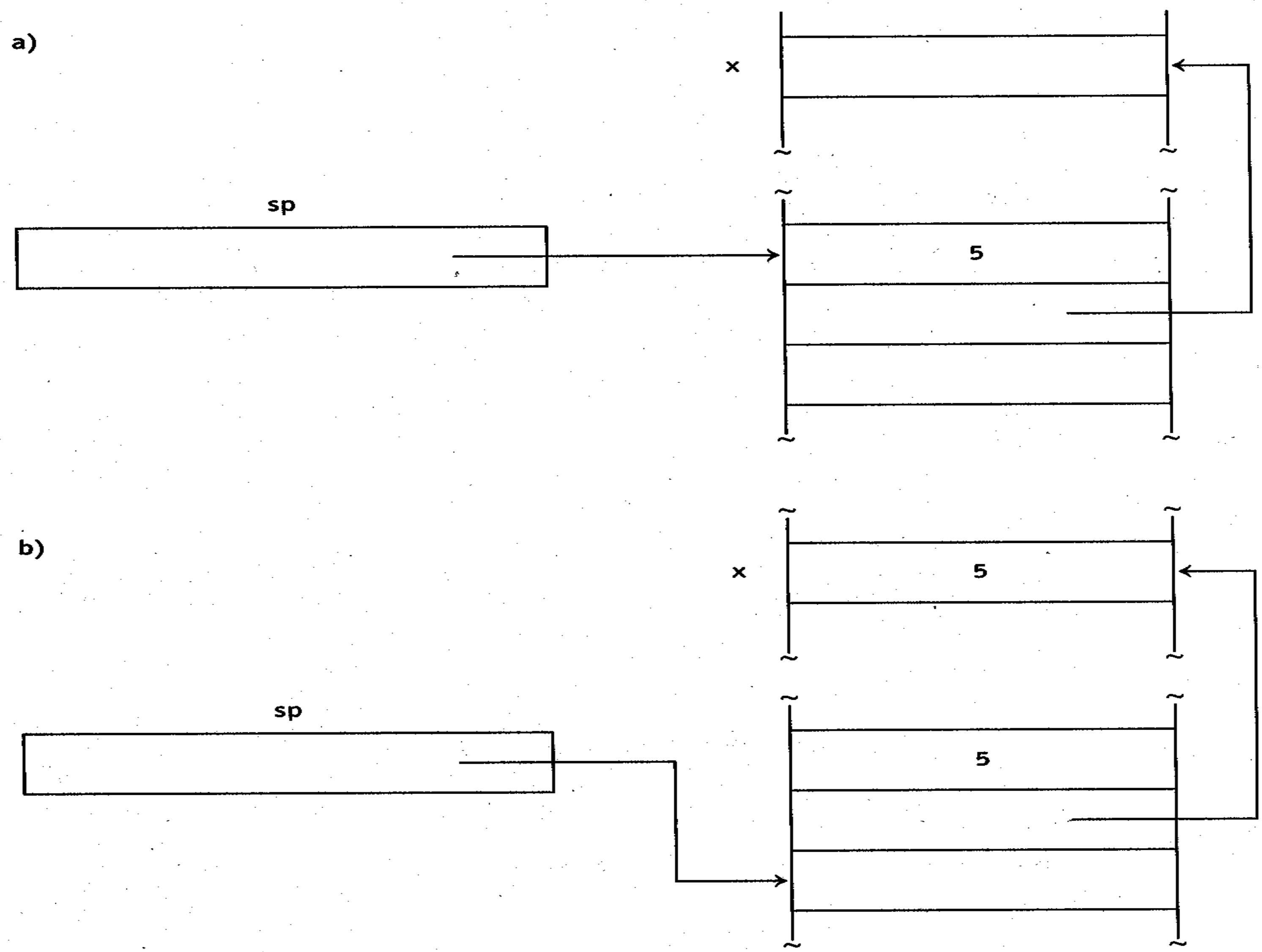
```
p 100      ; push value at location 100
p  x       ; push value at x
pr -2      ; push item at rel address 2
pc 5       ; push 5
pc x       ; push address of x
```

stav pops a value and address from stack in that order. Then it stores the value at the address.

See the next slide.

Effect of stav

FIGURE 13.35



stva is like stav except that it pops an address and value in that order.

y = x;

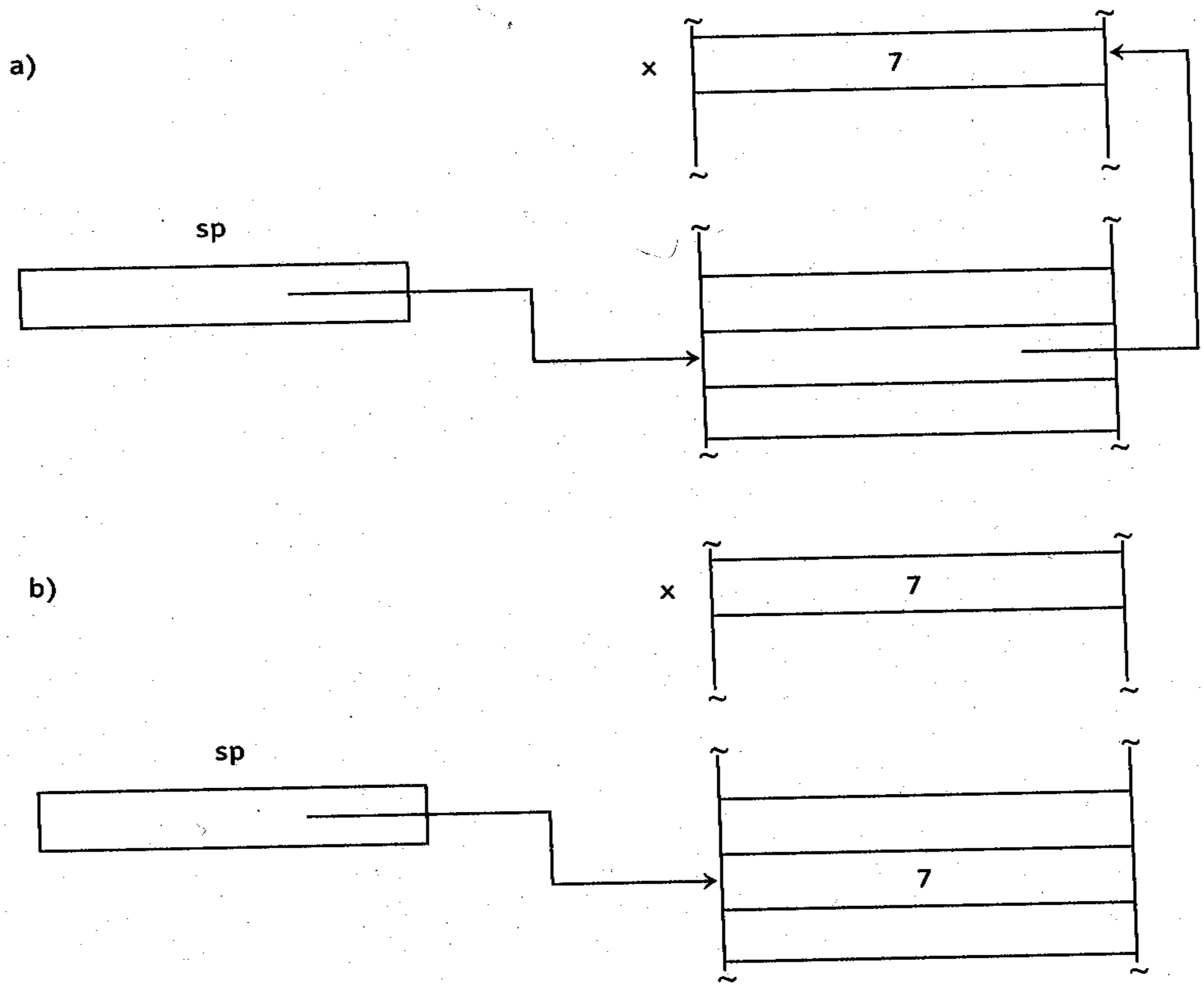
where x and y a global variables is translated to

```
pc y      ; push address ofy
p x       ; push value of x
stav
```

load pops a pointer and then pushes the value the pointer points to. **load** *dereferences* a pointer.
See the next slide.

Effect of load

FIGURE 13.36



y = *p;

where y and p are global variables is translated to

```
pc  y        ; push address of y
p   p        ; push value of p
load         ; get what p points to
stav
```

To compute 2 + 3:

```
pc 2
pc 3
add
```

add pops two values, adds them, and push the sum

To multiply 2 and 3:

```
pc 2
pc 3
mult
```

Arithmetic instructions do not use 4-bit opcodes so more 4-bit opcodes are available for other instructions.

To add two items with relative addresses -2 and -3:

```
pr  -2
pr  -3
add
```

One add instruction works for both absolute and relative addresses.

Conditional instructions (except for jcnt) pop and test the top of the stack.

See the next slide.

```
if (x < y)        // assume x and y are global signed ints
   x = 3;
```

is

```
     p    x          ; push value of x
     p    y          ; push value of y
     scmp            ; double pop and compare
     jzop @L1        ; jump if x >= y

     pc   x          ; x = 3;
     pc   3
     stav

@L1:
```

aspc in the stack instruction set can allocate and deallocate up to 4095 words:

```
aspc -2000   ; allocate 2000 words
```

cora in the stack instruction set contains the relative address to convert

cora -4   ; convert relative address 4

rev switches the top two stack items

```
pc   1
pc   2       ; 2 is on top of 1
```

2 is on top of the stack with 1 below it. If we then execute

```
rev          ; 1 is now on top of 2
```

Advantages of the stack instruction set

- Simpler compiling
- A more consistent instruction set because it does not require so many 4-bit opcodes. For example, add and mult work the same way (not true for the optimal instruction set).
- A more comprehensive set of instructions. For example, have a jump instruction for every possible condition.

Simpler compiling

x = a + b+ c;

optimal inst set	stack inst set
ld a	pc x
add b	p a
add c	p b
st x	p c
	stav

Compiler has to remember x in optimal instruction set.

x = a + b * c;

```
ld   b    ; compiler must remember x and a
push
ld   c
mult
add  a    ; use a here
st   x    ; use x here
```

In the stack instruction set, the compiler simply generates code as it scans the statement left to right. It does not have to remember **x** or **a**:

```
pc   x    ; push address of x
p    a    ; push a
p    b    ; push b
p    c    ; push c
mult      ; multiply b and c
add       ; add value of a
```

Function call that returns a value

```
sum = ret_sum(5, 7);
```

is

```
pc   sum          ; push address of sum
aspc -1           ; allocate return area
pc   7            ; create 2nd parameter
pc   5            ; create 1st parameter
call $ret_sum@ii
aspc 2            ; remove parameters
stav              ; place return value in sum
```

To return a value, the called function should leave the return value on the stack in a location provided by the caller.

```
cora   4        ; get address of ret area
pc     0        ; returning 0
stav            ; put 0 in return area
reba
ret
```

A complete program follows on the next two slides.

FIGURE 13.40

```
#include <iostream>
using namespace std;

int sum;
int ret_sum(int x, int y)
{
      int z;
      z = x + y;
      return z;
}
int main() {
      sum = ret_sum(5, 7);
      cout << sum << endl;
      return 0;
}
```

FIGURE 13.41

```
          !k              ; use the stack instruction set
@ret_sum$ii:
          esba

          aspc -1         ; int z;
                          ; z = x + y;
          cora -1              ; push address of z
          pr    2              ; push x
          pr    3              ; push y
          add                  ; compute x + y
          stav                 ; store sum in z
                          ; return z;
          pr   -1              ; push value of z
          cora  4              ; push address of return area
          stva                 ; save return value in return area
          reba
          ret
; ==========================================================
main:       esba
                          ; sum = ret_sum(5, 7);
            pc     sum        ; push address of sum
            aspc -1           ; allocate return area
            pc     7          ; create parameters
            pc     5
            call   @ret_sum$ii
            aspc   2          ; deallocate parameters
            stav              ; store return value in sum

            p      sum        ; cout << sum << endl;
            dout
            pc     '\n'
            aout
                              ; return 0;
            cora   4          ; push address of return area
            pc     0          ; push 0
            stav              ; store return value in return area
            reba
            ret
; ==========================================================
sum:         dw    0          ; int sum;
            public @ret_sum$ii
            public main
```

To run the program on the preceding slide, you must link it with start-up code for the stack instruction set (ksup.mob)

```
mas fig1341
lin fig1341 ksup
sim fig1341
```

Stack instruction set requires more memory accesses

```
ld   x    ; requires one memory access
add  y    ; requires one memory access
st   z    ; requires one memory access
```

6 total

which requires three memory accesses plus three more to access the three instructions. For the stack instruction set we have

```
pc   z    ; requires one memory access
p    x    ; requires two memory accesses
p    y    ; requires two memory accesses
add       ; requires three memory accesses
stav      ; requires three memory accesses
```

16 total

optimal instruction set: 37 microinstructions
stack instruction set: 77 microinstructions

FIGURE 13.42

```
Optimal Instruction Set

0: ld   /0 004/ ac=0000/0002   10t
1: add  /2 005/ ac=0002/0005  cy=0000/0000   17t
2: st   /1 006/ m[006]=0000/0005   8t
3: halt /FFFF /   2t

Stack Instruction Set

0: pc   /1 008/ m[FFF]=0000/0008  sp=0000/FFFF   10t
1: p    /0 006/ m[FFE]=0000/0002  sp=FFFF/FFFE   12t
2: p    /0 007/ m[FFD]=0000/0003  sp=FFFE/FFFD   12t
3: add  /F1 00/ m[FFE]=0002/0005  sp=FFFD/FFFE  cy=0000/0000  24t
4: stav /F3 00/ m[008]=0000/0005  sp=FFFE/0000   17t
5: halt /FFFF /   2t
```

The relatively poor performance of the stack instruction set may evaporate on simulated machines.

Thus, the choice of a stack-oriented instruction set for the JVM was not a bad choice.

Good projects to do now if you have already studied Chapter 6: write your own microcode for the optimal and stack instruction sets.

Use the file os.has for your optimal instruction set symbolic microcode. Test your microcode with osprog.mac (provided in the H1 Software Package)

```
has os
```

`has` will assemble `os.has` using the configuration file `os.cfg` in the H1 software package. Next, run the test program `osprog.mac` in the software package on `sim` by entering

```
sim osprog /z
```

The argument `/z` causes `osprog` to run with no trace. `osprog.mas` contains the !-directive `!os` so `sim` will automatically use your `os.hor` microcode. It will also use the `os.cfg` configuration file in the H1 software package. `sim` will respond with either

```
Simulator Version x.x
Reading configuration file os.cfg
Reading microcode file os.hor
No errors detected in optimal instruction set
```

if no errors are detected, or with

```
Simulator Version x.x
Reading configuration file os.cfg
Reading microcode file os.hor
ERROR detected in optimal instruction set at loc XXXX hex
```

Suppose the error address is 01D when you run osprog. Set a breakpoint at that address.

st at 1A is not working. So do over with breakpoint at 1A

```
---- [T7] 0: ldc  /8 000/ g
  0: ldc  /8 000/ ac=0000/0000
  1: swap /F7 00/ sp=0000/0000  ac=0000/0000
              .
              .
              .
 19: sub  /3 2F9/ ac=0002/0000
 1A: st   /1 308/
 1B: ld   /0 308/ ac=0000/0005
 1C: jz   /C 01E/
Machine-level breakpoint at 1D
---- [T7] 1D: call /E 288/
```

```
---- [T7] 1D: call /E 288/ o
Starting session.  Enter h or ? for help.
---- [T7] 0: ldc  /8 000/ b1a
Machine-level breakpoint set at 1A
---- [T7] 0: ldc  /8 000/ g
  0: ldc  /8 000/ ac=0000/0000
  1: swap /F7 00/ sp=0000/0000  ac=0000/0000
              .
              .
              .
 19: sub  /3 2F9/ ac=0002/0000
Machine-level breakpoint at 1A
---- [T7] 1A: st   /1 308/
```

Now enable **sim**, set the micro display mode, and trace one machine instruction. The trace then shows all the microlevel activity corresponding to the `st` instruction:

```
---- [T7] 1A: st   /1 308/ enable
Microlevel enabled
---- [T7] 1A: st   /1 308/ m
Microlevel display mode
---- [T1] 0: pc = 1 + pc; mar = pc; / ←hit ENTER
  0: pc = 1 + pc; mar = pc;
     mar=2F9/01A  pc=001A/001B
  1: rd;
     Rd from m[01A] mdr=0002/1308  1A: st   /1 308/
  2: ir = mdr; if (s) goto 7;
     ir=32F9/1308
  3: dc = left(ir); if (s) goto B;
     dc=97C8/2610
```

The microinstruction at 55 is missing wr, so assemble it in with the a command.

```
 4: dc = left(dc); if (s) goto 11;
    dc=2610/4C20
 5: dc = left(dc); if (s) goto 54;
    dc=4C20/9840
54: mdr = ac; mar = ir;
    mar=01A/308  mdr=1308/0000
55: goto 0;
---- [T1] 0: pc = 1 + pc; mar = pc; /
```

```
---- [T1] 0: pc = 1 + pc; mar = pc; / a55
55:  goto 0; / wr; goto 0
56:  mar = ir; /              ←hit ENTER to exit assembly mode
---- [T1] 0: pc = 1 + pc; mar = pc; /
```

Next we enter **o#** to do over. The **#** suffix on this command prevents **sim** from reinitializing microstore (which would overlay our fix):

```
---- [T1] 0: pc = 1 + pc; mar = pc; / o#
Starting session.  Enter h or ? for help.
---- [T1] 0: pc = 1 + pc; mar = pc; /
```

Finally, we enter **K** (in uppercase) to kill the previously set machine-level breakpoint, followed by **n** and **g** to see if **osprog** now succeeds:

```
---- [T1] 0: pc = 1 + pc; mar = pc; / K
Machine-level breakpoint killed
---- [T1] 0: pc = 1 + pc; mar = pc; / n
No display mode
---- [T7] g
No errors detected in optimal instruction set
```

Two-word instructions are possible. Maybe you can use them to create better instruction sets?

The o2 instruction set includes a two-word mult.

See sim.txt and o2.cfg to see how to set up a two-word instruction.

o2 better than o for *this* program

FIGURE 13.43 a)

```
1          !o
2          ld   x      ; 0005
3          push        ; F300
4          ld   y      ; 0006
5          mult        ; FF40
6          halt        ; FFFF
7 x:       dw   7      ; 0007
8 y:       dw   11     ; 000B
```

b)

```
1          !o2
2          ld   x      ; 0004
3          mult y      ; FF40 0005  (two-word instruction)
4          halt        ; FFFF
5 x:       dw   7      ; 0007
6 y:       dw   11     ; 000B
```